Linux Kernel Seminar Series: Interrupts and Exceptions

Jeffery A. Kuehn

Future Technologies / Scientific Computing

Oak Ridge National Laboratory

Overview

- Introduction to Interrupts and Exceptions
- The Role of Interrupts and Exceptions
- Interrupts & Exceptions
- Nested Execution of Exception and Interrupt Handlers
- Initializing the Interrupt Descriptor Table
- Exception Handling
- Interrupt Handling
- SoftIRQs, Tasklets, and Workqueues

Commonalities between Interrupts and Exceptions

- Generated by the hardware
  - vs. UNIX signals
- **Alter (divert) the sequence of instructions**
  - Diversion is similar to context switch (See chap 3)
  - Execution diverted to a “handler”
    - Not a process – lighter weight
    - Just a control path in the kernel
    - Executes from the diverted process's context
    - Think: handler == function
  - **Email/Telephone/Visitor analogy**

Differences: Exceptions

- Synchronous
  - Related to the diverted process
  - Internal to current thread of control
  - After the instruction terminates
  - May post a UNIX signal with the diverted process
- Think:
  - internal to CPU (mostly)
  - Device “nudges” itself

Exceptions: a closer look

- Come in several flavors
  - Processor Detected Exceptions
    - Faults
    - Traps
    - Aborts
  - Programmed Exceptions
- Each type of exception is assigned a “vector”
  - An 8 bit unsigned int (0-255) to identify it
- May send a UNIX signal to the diverted process

Processor Detected Exceptions: Faults

- Correction required
  - Ex: page fault must load a page
- Execution can continue
  - Saved EIP is instruction which caused fault
  - Program can resume execution at saved EIP
- Examples
  - Floating Point Error – divide by zero
  - General Protection Fault – one of the protection rules has been violated
  - Page Fault – more in Chapter 8

Processor Detected Exceptions: Traps

- Correction not required
- Execution can continue
  - Saved EIP is next instruction to execute
  - Mainly for debugging
- Example
  - **bound** – generate exception conditional on address bound
  - **into** – the instruction used to check for an integer overflow generates this exception if the overflow flag has been set

Processor Detected Exceptions: Aborts

- Correction not possible
- Execution cannot continue
  - EIP may be incorrect
  - Serious error, we're wedged
- Examples
  - Machine Check – CPU or bus hardware errors
  - Double fault – trying to handle the exception has generated an exception
  - Coprocessor segment overrun (386/387 only)

Programmed Exceptions

- Exceptions generated by special instructions
- Handled as traps (indistinguishable...)
  - Correction not required
  - Execution can continue
  - Saved EIP is next instruction to execute
- Examples:
  - **int** – system call
  - **int3** – breakpoint (debugger inserts)

Differences: Interrupts

- Asynchronous
  - Often unrelated to diverted process
- External to current thread of control
  - Ex: I/O device requesting attention
- arbitrary arrival time relative to instruction stream
- Think:
  - external to CPU (mostly)
  - One device “nudges” another for attention

Interrupts: A closer look

- Maskable Interrupts
  - Current execution state: masked or unmasked
  - CPU ignores until unmasked
  - examples:
    - Interrupt Requests (IRQs) issued by I/O devices
    - Timer Interrupts
    - Interprocessor Interrupts

- Non-Maskable Interrupts (NMIs)
  - Cannot be ignored
  - Critical hardware failures

- Each type of interrupt is assigned an 8 bit unsigned int (0-255)
  - This is called an interrupt "vector"
- Sitting at your desk:
  - Notification of email arriving is a maskable interrupt
  - TZ walking into your office is a non-maskable interrupt

There's a magic device....

- (Advanced) Programmable Interrupt Controller
- Monitors the IRQ lines on the bus for raised (electrical) signals
- When one or more IRQs are raised:
  - decides which will be handled first
  - Converts the IRQ to be serviced into an interrupt vector
  - Selects CPU to service the interrupt
  - Store vector in PIC I/O port where CPU can read it
  - Issues interrupt to selected CPU's interrupt line
    - ie. raise INTR pin
  - Waits for CPU to acknowledge by writing into one of the PIC's I/O ports
  - Clears interrupt line

Old Magic: PICs – Intel 8259A

- Could monitor 8 IRQ lines
  - Two cascaded could monitor 15 (orig PC)
- If multiple interrupts posted simultaneously
  - lowest pin number was highest priority and handled first
- Default vector association for IRQn was n+32 – could be modified
- Each IRQ can be selectively enabled/disabled
  - This is not global masking/unmasking
    - When masked, PIC still issues interrupts
    - CPU just ignores them temporarily
  - Disabled interrupts are not lost
    - Passed to CPU when re-enabled
  - Used by handlers to allow serial processing of interrupts of the same type
- Okay for single CPU, but not SMPs

New Magic: PICs for SMPs

- Local APIC (LAPIC) built into each CPU
- I/O APIC for each external I/O bus
  - System can have many I/O APICs
- APICs connected via Interrupt Controller Communication (ICC) bus
  - Mostly invisible to software

New magic: Local APIC

- Built into CPU
- 32 bit registers
  - Task Priority Register (TPR)
    - set by OS during process switch
- Clock
- Timer
- 2 additional local IRQ lines
  - Reserved for local interrupts

New magic: External I/O APIC

- Programmable Registers & Message Unit
- 24 IRQ lines
- 24-entry Interrupt Redirection Table
  - translate IRQ line to an ICC bus message to one or more LAPICS
  - Each table entry contains
    - Interrupt Vector
    - *Programmable priority – not tied to pin number*
    - *Programmable service processor selection method*
    - *Programmable Destinations*
  - Translates IRQ line into a message to one or more LAPICS

Service processor selection method details for I/O Interrupts

- Static distribution
  - specific CPU
  - Specific Subset of CPUs
  - all CPUs at once (bcast)
- Dynamic distribution
  - Check current TPR values on LAPICs
  - Assign service to CPU executing lowest priority task (+arbitration)

Interrupt Descriptor Table

- System table
  - 256 descriptor entries
  - 2048 bytes total
  - **idtr** register holds base/limit
  - **idtr initialized by lidt instruction**
- Three types of descriptor entries
  - 8 bytes (64 bits) each
  - Intel Task gate descriptor
  - Intel Interrupt gate descriptor
  - Intel Trap gate descriptor
- Associates each interrupt or exception vector with address of corresponding handler

Intel Task Gate Descriptor

- 4 bit Type field
- 2 bit Descriptor Privilege Level (DPL) field
- 16 bit Task State Segment (TSS) selector of the process which will take control when an interrupt occurs
- Used to deliver “Double Fault” interrupts

Intel Interrupt Gate Descriptor

- 4 bit Type field
- 2 bit Descriptor Privilege Level (DPL) field
- 48 bit Address of handler
  - segment selector + offset
- Clears **IF** flag
  - disables maskable interrupts while transferring control
- Used for most interrupt handlers

Intel Trap Gate Descriptors

- 4 bit Type field
- 2 bit Descriptor Privilege Level (DPL) field
- 48 bit Address of handler
  - segment selector + offset
  - like interrupt gate
- Doesn't modify IF flag
- Used for exception handlers

How Linux Uses the Intel Gates(1)

- (Linux) Task Gate
  - Activates “Double Fault” handler
  - Privileged (Intel) Task gate (DPL=0 kernel mode)
  - set_task_gate(n,GDT)
- (Linux) Interrupt Gate
  - Activates all Linux interrupt handlers
  - Privileged (Intel) Interrupt gate (DPL=0 kernel mode)
  - set_intr_gate(n,addr)
- (Linux) System Interrupt Gate
  - Activates Linux exception handler for **int3** instruction (breakpoint – vector 3)
  - Unprivileged (Intel) interrupt gate (DPL=3 user mode)
  - set_system_intr_gate(n,addr)

How Linux Uses the Intel Gates(2)

- (Linux) Trap Gate
  - Activates most Linux exception handlers
  - Privileged (Intel) trap gate (DPL=0 kernel mode)
  - set_trap_gate(n,addr)
- (Linux) System Gate
  - Activates Linux exception handlers for 3 instructions
    - **into** (overflow check – vector 4)
    - **bound (address check – vector 5)**
    - **int $0x80** (system call – vector 128)
  - Unprivileged (Intel) trap gate (DPL=3 user mode)
  - set_system_gate(n,addr)

Nested Execution of Handlers(1)

- Kernel control paths can be arbitrarily nested
  - Exceptions can go 2 levels deep
    - User – some exception – page fault
  - Interrupts can go arbitrarily deep
    - User – device1 – device2 – device3 ...
  - Interrupt handlers can preempt exception handlers and other interrupt handlers
    - User – some exception – intr1 – intr2 – intr3 ...
  - Exception handlers cannot preempt interrupt handlers (common problem...)
- Why?
  - Improve PIC service throughput (faster)
  - Eliminates need for priority levels (simpler)

Nested Execution of Handlers(2)

- Restrictions:
  - Can be entered from user mode or kernel mode
  - Must return to previous
  - First task of any handler is save old context
  - Last task of any handler is restore old context
  - Handler must never block
    - No process switches
    - Handler must not attempt I/O or other blocking operations
    - Not allowed to induce a page fault
      - (which terminates with a process switch)
    - Remember: interrupts are disabled!

Structure of an Exception Handler

– Assembler Wrapper – low level handler
  - Named “*handler_name*”
  - Saves the previous context on entry
    – Including switching stacks if necessary
  - Sets up the current context for a C call
  - Calls the high level C handler function
  - Cleans up the current context after C call
  - Restores previous context
    – Including switching stacks if necessary
  - returns to previous context
– C function – high level handler
  - named “do_*handler_name*”

do_*handler_name*()

- Std C function except args passed in registers
- Does all the heavy lifting
  - Always call notify_die()
    - To check whether exception occurred in kernel mode
    - Invalid system call arguments (chapter 10)
    - Kernel bugs
      - Invoke **die()** - prints CPU regs to console (kernel oops)
      - Invoke **do_exit()** - terminates current process
  - Typically calls **do_trap()** to
    - Store HW error code/exception vector
    - Send a (UNIX) signal to the process
      - Handled immediately after exception handler terminates
  - Returns (to wrapper)
- See linux/arch/i386/kernel/traps.c

do_*handler_name*()

```
#define DO_VM86_ERROR(trapnr, signr, str, name) \
fastcall void do_##name(struct pt_regs * regs, long error_code) \
{ \
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) == NOTIFY_STOP) return; \
do_trap(trapnr, signr, str, 1, regs, error_code, NULL); \
}


#ifndef CONFIG_KPROBES
DO_VM86_ERROR( 3, SIGTRAP, "int3", int3)
#endif
```

do_*handler_name*()

```
#define DO_ERROR(trapnr, signr, str, name) \
fastcall void do_##name(struct pt_regs * regs, long error_code) \
{ \
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) == NOTIFY_STOP) return; \
do_trap(trapnr, signr, str, 0, regs, error_code, NULL); \
}
```

```
DO_ERROR( 9, SIGFPE,  "coprocessor segment overrun", coprocessor_segment_overrun)
DO_ERROR(10, SIGSEGV, "invalid TSS", invalid_TSS)
DO_ERROR(11, SIGBUS,  "segment not present", segment_not_present)
DO_ERROR(12, SIGBUS,  "stack segment", stack_segment)
```

do_*handler_name*()

```
#define DO_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr) \
fastcall void do_##name(struct pt_regs * regs, long error_code) \
{ \
siginfo_t info; \
info.si_signo = signr; \
info.si_errno = 0; \
info.si_code = sicode; \
info.si_addr = (void __user *)siaddr; \
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) == NOTIFY_STOP) return; \
do_trap(trapnr, signr, str, 0, regs, error_code, &info); \
}

DO_ERROR_INFO( 6, SIGILL,  "invalid opcode", invalid_op, ILL_ILLOPN, regs->eip)
DO_ERROR_INFO(17, SIGBUS, "alignment check", alignment_check, BUS_ADRALN, 0)
DO_ERROR_INFO(32, SIGSEGV, "iret exception", iret_error, ILL_BADSTK, 0)
```

do_*handler_name*()

```
#define DO_VM86_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr) \
fastcall void do_##name(struct pt_regs * regs, long error_code) \
{ \
siginfo_t info; \
info.si_signo = signr; \
info.si_errno = 0; \
info.si_code = sicode; \
info.si_addr = (void __user *)siaddr; \
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) == NOTIFY_STOP) return; \
do_trap(trapnr, signr, str, 1, regs, error_code, &info); \
}

DO_VM86_ERROR_INFO( 0, SIGFPE,  "divide error", divide_error, FPE_INTDIV, regs->eip)
```

Exception Wrappers

- See linux/arch/kernel/i386/kernel/entry.S

```
ENTRY(overflow)
    pushl $0
    pushl $do_overflow
    jmp error_code
ENTRY(bounds)
    pushl $0
    pushl $do_bounds
    jmp error_code
ENTRY(invalid_op)
    pushl $0
    pushl $do_invalid_op
    jmp error_code
  ...
```

error_code – more assembler

- Same for all exceptions
  - 35 lines of assembler
  - Includes 2 function calls
    - Stack fixer
    - C handler function
- Tasks:
  - Save the context
  - Point **esp** to the right stack
  - Copies HW error code (if any) onto the stack
  - calls C handler function whose address is on the stack
  - When it returns it jumps to ret_from_exception

arch/kernel/i386/kernel/entry.S:error_code

```
error_code:
    pushl %ds
    pushl %eax
    xorl %eax, %eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    decl %eax          # eax = -1
    pushl %ecx
    pushl %ebx
    cld
    pushl %es
    UNWIND_ESPFIX_STACK
    popl %ecx
    movl ES(%esp), %edi   # get the function address
    movl ORIG_EAX(%esp), %edx     # get the error code
    movl %eax, ORIG_EAX(%esp)
    movl %ecx, ES(%esp)
    movl $(__USER_DS), %ecx
    movl %ecx, %ds
    movl %ecx, %es
    movl %esp,%eax    # pt_regs pointer
    call *%edi
    jmp ret_from_exception
```

ret_from_exception

- Same for all exceptions (& interrupts)
- A little more involved, but...
- Tasks:
  - Clean up the stack
  - Point esp to the right stack
  - If previous context was user mode
    - Restore previous (user) context and return
  - Else, previous context was kernel mode
    - If preemption is enabled
      - need_resched? --> call preempt_schedule_irq
    - Otherwise restore previous (kernel) context and return

ret_from_exception

```
/* Return to user mode is not as complex as all this
 * looks, but we want the default path for a system
 * call return to go as quickly as possible which is
 * why some of this is less clear than it otherwise
 * should be. */
        ALIGN   # userspace resumption stub
                # bypassing syscall exit tracing
ret_from_exception:
        preempt_stop
ret_from_intr:
        GET_THREAD_INFO(%ebp)
        movl EFLAGS(%esp), %eax                 #
                mix EFLAGS and CS
        movb CS(%esp), %al
        testl $(VM_MASK | 3), %eax
        jz resume_kernel
ENTRY(resume_userspace)
        cli     # make sure we don't miss an
                # interrupt setting need_resched or
                # sigpending  between sampling
                # and the iret
        movl TI_flags(%ebp), %ecx
        andl $_TIF_WORK_MASK, %ecx  # is there
            any work to be done on  int/exception
            return?
        jne work_pending
        jmp restore_all

#ifdef CONFIG_PREEMPT
ENTRY(resume_kernel)
        cli
        cmpl $0,TI_preempt_count(%ebp) # non-
            zero preempt_count ?
        jnz restore_nocheck
need_resched:
        movl TI_flags(%ebp), %ecx   # need_resched
            set ?
        testb $_TIF_NEED_RESCHED, %cl
        jz restore_all
        testl $IF_MASK,EFLAGS(%esp)     #
            interrupts off (exception path) ?
        jz restore_all
        call preempt_schedule_irq
        jmp need_resched
#endif
```

Interrupt handling vs Exception handling

- Exceptions
  - Current process is responsible
  - Mostly handled by posting a UNIX signal
  - Defers action until signal is received
  - Fast
- Interrupts
  - Current process is probably unrelated
  - Responsible process was likely suspended long ago

Three types of Interrupts

- Three main types of interrupts
  - Interprocessor interrupts
    - Simple. Discuss first.
  - I/O interrupts
    - Complex. Discuss second.
  - Timer interrupts
    - Specialized. Discuss in Chapter 6.
- Each will require different handling approaches

Structure of an Interprocessor Interrupt Handler

- Three kinds of interprocessor interrupts:
  - CALL_FUNCTION_VECTOR (vector 0xfb)
    - Force target CPUs to run function specified by sender
  - RESCHEDULE_VECTOR (vector 0xfc)
    - Forces target to rerun scheduler
  - INVALIDATE_TLB_VECTOR (vector 0xfd)
    - Forces target to invalidate their TLB

Interprocessor Interrupts

- Handled by LAPIC
  - Write vector and target id to own LAPIC's Interrupt Command Register (ICR)
  - Sends message across ICC bus to target CPU's LAPIC
  - Target CPU's LAPIC issues interrupt to its CPU

IPI Handler Tasks

- Enter assembly wrapper (aka low level handler)

  call_function_interrupt:

  reschedule_interrupt:

  invalidate_tlb_interrupt:

- Save context
- Push *vector*-256 onto stack
- Call C function handler (aka high level handler)

  smp_call_function_interrupt()

  smp_reschedule_interrupt()

  smp_invalidate_tlb_interrupt()

  – Acknowledge the interrupt
  – Performs the requested action
    - Note that **smp_reschedule_interrupt()** does nothing here – the resched is a side effect of exiting the handler!
  – Returns to wrapper (low level handler)
- Restores previous context and returns... as seen on TV
  – Identical to exceptions except for that extra **cli** instruction

Structure of an I/O Interrupt Handler

- Handler breaks tasks into three classes
  - Critical tasks
    - Acknowledging Interrupt to PIC
    - Reprogramming PIC or device controller
    - Updating data structures shared by processor & device
    - Fast tasks like writing the ack to the PIC
    - Performed in the "top half" with maskable interrupt disabled
  - Non-critical tasks
    - Updating data structures used only by the processor
    - Fast tasks like looking up a keyscan code
    - Setting up a softirq or tasklet to handle deferred tasks
    - Performed in the "top half" with maskable interrupts enabled
  - Non-critical Deferable tasks
    - Slow tasks like copying buffers
    - Performed (much) later as a softirq or tasklet

Four basic actions of an I/O Interrupt Handler (cont)

- Save previous context
- Acknowledge the interrupt to the PIC
  - Allows PIC to issue further interrupts
- Call ***ALL*** interrupt service routines (ISRs) associated with the devices which share the IRQ
- Terminate and return to previous context

I/O Interrupt Handler Implementation

- Back to linux/arch/i386/kernel/entry.S
- Entry: **irq_entries_start:**
  - Push IRQ number onto stack
  - Jump to **common_interrupt:**
- **common_interrupt:**
  - Save the context
  - copy IRQ number into **eax**
  - Call **do_IRQ()**
    - Executes all ISRs associated with the interrupt
  - Call **ret_from_intr:** when **do_IRQ()** returns

Interrupt entry code

```
/* Build the entry stubs and pointer
 * table with some assembler magic.*/
.data
ENTRY(interrupt)
.text
vector=0
ENTRY(irq_entries_start)
.rept NR_IRQS
	ALIGN
1:	pushl $vector-256
	jmp common_interrupt
.data
	.long 1b
.text
	vector=vector+1
.endr
	ALIGN
common_interrupt:
	SAVE_ALL
	movl %esp,%eax
	call do_IRQ
	jmp ret_from_intr
```

do_IRQ() actions:

- **irq_enter()** macro (linux/include/linux/hardirq.h)
  - Account for system time
  - increments nesting count in thread_info of current proc
- Call **__do_IRQ()**
- **irq_exit()** macro
  - Decrements nesting count
  - If not already in interrupt context, handle any pending softirqs
  - return

kernel/irq/handle.c:__do_IRQ() Tasks

- If there's a local interrupt
  - call its handler first
  - Locking not required
- Lock IRQ descriptor
- Acknowledge the interrupt
- Mark it pending
- If we can handle it now
  - mark it in_progress
  - If not, it's still pending
  - & someone will git-r-done

- Loop:
  - Fire off the “non-critical” fast work
    - Unlock IRQ descriptor
    - Call **handle_IRQ_event()**
    - Lock IRQ descriptor
  - Check for another event on this IRQ
  - If there's not another, break out
  - Clear the pending flag
  - Back to top
- Call the end() handler to deal with disabled interrupts arriving while this handler was running
- Unlock IRQ descriptor
- return

handle_IRQ_event()

- Trivial: walk the linked list containing pointers to handler functions and dispatch them
- Local IRQs are enabled while the handlers run
- The action is part of the device driver and will be discussed later, but it may help to glance at a simple example...

```
fastcall int handle_IRQ_event(unsigned int irq, struct pt_regs
      *regs, struct irqaction *action)
{
      int ret, retval = 0, status = 0;
      if (!(action->flags & SA_INTERRUPT))
              local_irq_enable();
      do {
              ret = action->handler(irq, action->dev_id, regs);
              if (ret == IRQ_HANDLED) status |= action->flags;
              retval |= ret;
              action = action->next;
      } while (action);
      if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
      local_irq_disable();
      return retval;
}
```

Action example: floppy handler

```
irqreturn_t floppy_interrupt(int irq, void *dev_id, struct pt_regs *regs)  {
    void (*handler) (void) = do_floppy;
    int do_print;
    unsigned long f;
    lasthandler = handler;
    interruptjiffies = jiffies;
    f = claim_dma_lock();
    fd_disable_dma();
    release_dma_lock(f);
    floppy_enable_hlt();
    do_floppy = NULL;
    if (fdc >= N_FDC || FDCS->address == -1) {
        /* we don't even know which FDC is the culprit */
        printk("DOR0=%x\n", fdc_state[0].dor);
        printk("floppy interrupt on bizarre fdc %d\n", fdc);
        printk("handler=%p\n", handler);
        is_alive("bizarre fdc");
        return IRQ_NONE;
    }
FDCS->reset = 0;
do_print = !handler && print_unex && !initialising;
    inr = result();
    if (do_print)  print_result("unexpected interrupt", inr);
    if (inr == 0) {
        int max_sensei = 4;
        do {
            output_byte(FD_SENSEI);
            inr = result();
            if (do_print)   print_result("sensei", inr);
            max_sensei--;
        } while ((ST0 & 0x83) != UNIT(current_drive) && inr == 2
     && max_sensei);
    }
    if (!handler) {
        FDCS->reset = 1;
        return IRQ_NONE;
    }
    schedule_bh(handler);
    is_alive("normal interrupt end");
    return IRQ_HANDLED;
}
```

Action example: floppy handler

- Does 2.6 really have a top half / bottom half architecture?
  - No, not like older kernels (-2.4)
  - But yes, the driver is still split
  - Everything in the interrupt call tree above **do_floppy()** is “top half” ...
  - **do_floppy()** points to the “bottom half” command
- **do_floppy()** points to the most recently active command in the driver
- different commands (functions) in the floppy device driver code:
  - **do_floppy = main_command_interrupt;**
  - **do_floppy = seek_interrupt;**
  - **do_floppy = recal_interrupt;**
  - **do_floppy = reset_interrupt;**
- **schedule_bh()**
  - Now uses **schedule_work()** to place Bottom Half (BH) on a workqueue for later execution
  - linux/kernel/workqueue.c

Action example: final look

```
irqreturn_t floppy_interrupt(int irq, void *dev_id, struct pt_regs *regs)  {
    void (*handler) (void) = do_floppy;
    int do_print;
    unsigned long f;
    lasthandler = handler;
    interruptjiffies = jiffies;
    f = claim_dma_lock();
    fd_disable_dma();
    release_dma_lock(f);
    floppy_enable_hlt();
    do_floppy = NULL;
    if (fdc >= N_FDC || FDCS->address == -1) {
        /* we don't even know which FDC is the culprit */
        printk("DOR0=%x\n", fdc_state[0].dor);
        printk("floppy interrupt on bizarre fdc %d\n", fdc);
        printk("handler=%p\n", handler);
        is_alive("bizarre fdc");
        return IRQ_NONE;
    }
FDCS->reset = 0;
do_print = !handler && print_unex && !initialising;
    inr = result();
    if (do_print)  print_result("unexpected interrupt", inr);
    if (inr == 0) {
        int max_sensei = 4;
        do {
            output_byte(FD_SENSEI);
            inr = result();
            if (do_print)   print_result("sensei", inr);
            max_sensei--;
        } while ((ST0 & 0x83) != UNIT(current_drive) && inr == 2 &&
max_sensei);
    }
    if (!handler) {
        FDCS->reset = 1;
        return IRQ_NONE;
    }
    schedule_bh(handler);
    is_alive("normal interrupt end");
    return IRQ_HANDLED;
}
```

1) Pick up the outstanding command from a static pointer
2) Reset the device
3) Clear the static command pointer
4) Schedule the command on a workqueue for later execution
5) Start returning back up the chain – top half complete!

SoftIRQs, Tasklets, and Workqueues

- Top half
  - Must be fast
  - Not deferrable
  - Acknowledge the interrupt
  - Schedule the “real” work
  - Return to the user process
- Bottom half
  - Can be slow
  - Deferrable
  - Do the “real” work
    - At a convenient time
- Three methods for deferral
  - Softirqs – appeared in 2.4 kernel
    - No serialization
    - Fastest
    - All softirqs (even same type) run concurrently
  - Tasklets – appeared in 2.4 kernel
    - Nothing to do with “tasks”
    - Simpler interface to softirqs
    - Based on softirqs
    - Different types of tasklets run in parallel
  - Workqueues – new in 2.6 kernels
    - Highest overhead
    - Run in process context – can sleep
    - Easiest to use

softirqs

- See linux/kernel/softirq.c
- Statically allocated (32) at compile time
- Can run concurrently on multiple CPUs
  – Even same the same type
- Must be re-entrant
  – Must protect data structures from concurrent access with spinlocks (expensive)
- Six types

• HI_SOFTIRQ	0 – high priority tasklets
• TIMER_SOFTIRQ	1 – tasklets related to timers
• NET_TX_SOFTIRQ	2 – transmit packets to network
• NET_RX_SOFTIRQ	3 – receive packets from network
• SCSI_SOFTIRQ	4 – post-processing SCSI commands
• TASKLET_SOFTIRQ	5 – regular tasklets

Tasklets

- See linux/kernel/softirq.c
- Dynamically allocated during runtime
  - Module load
- Tasklets of different types can execute concurrently on multiple CPUs
- Tasklets of the same type are serialized
- Tasklets need not be re-entrant or protect their data structures
- Implemented using softirqs

Operations on softirqs and tasklets

- Initialization
  - Define a new deferrable function, as during module load
- Activation
  - Mark a deferrable function as “pending”
  - Execute next time kernel schedules deferrables
  - Can be done any time, often by top half of interrupt handler
- Masking
  - Selectively disable a deferrable to prevent execution even if activated (chapter 5)
- Execution
  - Executes pending deferrable with other pending deferrables at “well-specified” times
  - Will execute on the same CPU that activated it

Softirq key data structures

```
/* linux/include/linux/interrupt.h
struct softirq_action {
        void    (*action)(struct softirq_action *); /* function ptr */
        void    *data;
};

/* linux/kernel/softirq.c */
static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;
```

- A table of 32 structures containing pairs of function and data pointers
  - note only the first 6 are used
  - softirq priority (0-5) is index into table
  - HI_SOFTIRQ 0 – high priority tasklets
  - TIMER_SOFTIRQ 1 – tasklets related to timers
  - NET_TX_SOFTIRQ 2 – transmit packets to network
  - NET_RX_SOFTIRQ 3 – receive packets from network
  - SCSI_SOFTIRQ 4 – post-processing SCSI commands
  - TASKLET_SOFTIRQ 5 – regular tasklets

Softirq key data structures (2)

- **Recall that the handling of an interrupt can be interrupted by an interrupt. How deep are we nested?**
- **current_thread_info()->preempt_count** field
  - **preemption** counter (8bits)
    - how many times have we disabled preemption on this CPU?
    - 0 == preemption not explicitly disabled
  - **softirq** counter (8bits)
    - how many levels of deferral deep have we disabled
    - 0 == deferrable functions enabled
  - **hardirq** counter (12 bits)
  - **PREEMPT_ACTIVE** flag (1bit)
- Last two checked by **in_interrupt()** macro
  - Returns zero if both fields are zero

Softirq key data structures (3)

- Struct irq_cpustat_t's **__softirq_pending** field
  - 32 bit mask of pending interrupts
  - One struct per cpu
  - Accessed by **local_softirq_pending()** macro
    - Selects mask for local CPU

Activating softirqs with raise_softirq()

1) Save state and disable interrupts on local CPU
   – Performed by **local_irq_save()** macro
2) Mark softirq as pending
   – Set bit in **__softirq_pending mask**
3) If **in_interrupt()** returns 1 skip to step 5
   – Indicates either
      • Softirqs are currently disabled, or
      • **raise_softirq()** has been called in an interrupt context
4) If necessary, wake up local CPU's **ksoftirqd** kernel thread
   – **wakeup_softirqd()**
   – Restore state
   – Performed by **local_irq_restore()** macro

Pending softirqs

- Kernel checks for pending softirqs periodically
  - When **local_bh_enable()** is invoked
  - When **do_IRQ()** finishes and invokes **irq_exit()**
  - When **smp_apic_timer_interrupt()** finishes
    - Only if we have an APIC
  - When SMP CPU finishes function triggered by **CALL_FUNCTION_VECTOR** IPI
  - When one of the **ksoftirqd**/***n*** threads wakes up

do_softirq() (finally!)

- Invoked to handle pending softirqs noted at checkpoints on previous slide
- tasks:
  - Give up if **in_interrupt()** returns 1
    - Either softirqs are disabled or one is already in progress
  - Save state and disable interrupts with **local_irq_save()**
  - Switch to interrupt stack (if necessary)
  - Call **__do_softirq()**
  - Restore stack (if necessary)
  - Restore state and interrupts with **local_irq_restore()**
  - return

__do_softirq() (arrrgh!)

- Reads the softirq bitmask on the local CPU
  - Each set bit corresponds to pending softirqs
  - Executes the deferrables associated with every set bit
- New softirqs can be posted as pending during processing
  - Loop to catch them
  - Limited to 10 trips to avoid monopolizing the CPU
  - User mode processes locked out during handling
- If there are still pending softirq's after the 10th trip, we wake up the ksoftirqd for this CPU
  - Competes with user mode processes at low priority

__do_softirq() tasks

1) Initialize iteration counter to 10
2) Copy softirq bitmask of local CPU into local variable **pending**
3) Call **local_bh_disable()** [must execute serially]
4) Clear softirq bitmask of the local CPU so we can see if new softirqs arrive while we're working
5) Enable interrupts locally with **local_irq_enable()**
6) For every set bit in **pending**, execute the function **softirq_vec[n]->action()**
7) Disable interrupts locally with **local_irq_disable()**
8) Copy the softirq bitmask into **pending** again
9) If **pending** is non-zero and we haven't exceeded our iteration count, jump back to step 4
10) If there are more pending softirqs, call **wakeup_softirqd()**
11) Call **local_bh_enable()**

ksoftirqd/n

- Kernel thread running at low priority
- One per CPU
- Calls **do_softirq()** while there are pending softirqs
- Competes with user mode processes, but doesn' t lock them out

Tasklets

- Built on two softirqs
  - **HI_SOFTIRQ**
  - **TASKLET_SOFTIRQ**
  - Only difference is priority (**HI_SOFTIRQ** runs first)
- Each softirq can have several tasklets
- Tasklets are stored in two vectors:
  - **tasklet_vec[NR_CPUS]**
  - **tasklet_hi_vec[NR_CPUS]**
  - Each element of the vectors is the head of a linked list of tasklet descriptors
  - **do_irq()** calls **tasklet_hi_action()** and **tasklet_action()**
    - Remember: **__do_softirq()** really does the call...
    - **tasklet_hi_action()** and **tasklet_action()** handle the serialization

Workqueues

- Unlike deferrable functions which run in an interrupt context, workqueue tasks always run in the process context of a kernel thread
- Like tasklets, the core of the workqueue is a linked list of functions to be executed
- Unlike deferrable functions, queuetasks can block
  - But it holds up the queue...
- No access to a usermode address space

“jmp ret_from_intr”